A Model for Data Processing Programs

– Technical Report –

João Batista de Souza Neto[1], Anamaria Martins Moreira[2], Genoveva Vargas-Solar[3], and Martin A. Musicante[1]

[1] Department of Informatics and Applied Mathematics (DIMAp)
Federal University of Rio Grande do Norte, Natal, Brazil.
jbsneto@ppgsc.ufrn.br, mam@dimap.ufrn.br
[2] Computer Science Department (DCC)
Federal University of Rio de Janeiro, Rio de Janeiro, Brazil.
anamaria@dcc.ufrj.br
[3] Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG-LAFMIA, Grenoble, France.
genoveva.vargas@imag.fr

1 Introduction

In this report we describe a model to represent programs for large scale data processing. This model was designed based on common features in such programs, such as those on the systems *Apache Spark* [10], *DryadLINQ* [9], *Apache Beam* [1] and *Apache Flink* [2]. These systems have similarities in the way in which a program is represented and the operations on datasets that can be performed. Based on these similarities, we define a model that covers the main functional characteristics of data processing programs. In the mentioned systems, a program is represented as a *directed acyclic graph* (DAG) in which its vertices represent operations on datasets. These operations, in turn, define manipulations and processing in the datasets. Thus, a program is defined as a DAG composed of successive transformations in the dataset that is being processed. Our model represents these two aspects, DAG and operations, separately. We refer to *data flow* as the representation of the program's DAG and *transformations* as the operations applied to that program. We use *Petri Nets* [7], a formalism for distributed systems, to represent the data flow, and *Monoid Algebra* [4], an algebra for processing distributed data, to define the transformations.

We start with an introduction to Petri Nets and Monoid Algebra that underlie our model. Then, we present the definition of the data flow and finish with the definition of the transformations.

2 Background

This section briefly presents Petri Nets and Monoid Algebra, upon which our model is built. For a more detailed presentation, the reader can refer to [7, 4].

2.1 Petri Nets

Petri Nets [8] is a formal tool that allows to model and analyze the behavior of distributed, concurrent, asynchronous, and/or non-deterministic systems [7]. A Petri Net is defined as a directed bipartite graph that contains two types of nodes: *places* and *transitions*. Places represent the system's state variables, while transitions represent the actions performed by the system. These two components are connected through directed edges that connect places to transitions and transitions to places. With these three components it is possible to represent the different states of a system (places), the actions that take the system from one state to another (transitions) and how the change of state is made due to actions (edges). This is done by using *tokens* to decorate places of the net. The distribution of the tokens among places indicates that the system is in a given state. The execution of an action (transition) takes tokens from one state (place) to another.

Formally, a Petri net is a quintuple $PN = (P, T, F, W, M_0)$ where $P \cap T = \emptyset$, $P \cup T \neq \emptyset$ and:

$$\begin{aligned}
P = \{p_1, p_2, \ldots, p_m\} &\quad \text{is a finite set of } \textit{places},\\
T = \{t_1, t_2, \ldots, t_n\} &\quad \text{is a finite set of } \textit{transitions},\\
F \subseteq (P \times T) \cup (T \times P) &\quad \text{is a finite set of } \textit{edges},\\
W : F \to \{1, 2, 3, \ldots\} &\quad \text{is a set of weights associated to edges},\\
M_0 : P \to \{0, 1, 2, 3, \ldots\} &\quad \text{is a function defining the initial marking of a net},
\end{aligned}$$

The execution of a system is defined by *firing* transitions. Firing a transition t consumes $W(s,t)$ tokens from all its input places s, and produces $W(t,s')$ tokens to each of its output places s'.

2.2 Monoid Algebra

Monoid Algebra was proposed in [4] as an algebraic formalism for data-centric distributed computing operations based on monoids and monoid homomorphisms. A monoid is a triad $(S, \oplus, e_\oplus)$ formed by a set S, an associative operation $\oplus$ in S and a neutral element $e_\oplus$ (consider that $\oplus$ identifies the monoid). A monoid homomorphism is an H function over two monoids, from $\otimes$ to $\oplus$, which respects:

$$\begin{aligned}
H(X \otimes Y) &= H(X) \oplus H(Y) \quad \text{for all } X \text{ and } Y \text{ of type } S\\
H(e_\otimes) &= e_\oplus
\end{aligned}$$

Monoid algebra uses the concepts of monoid and monoid homomorphism to define operations on distributed datasets, which are represented as monoid collections. One type of monoid collection is the *bag*, an unordered data collection of type α (denoted as $Bag[\alpha]$) that has the unit injection function $\mathbb{U}_\uplus$, which generates the bag $\{\!\{x\}\!\}$ from the unitary element x ($\mathbb{U}_\uplus(x) = \{\!\{x\}\!\}$), the associative operation $\uplus$, which unites two bags ($\{\!\{x\}\!\} \uplus \{\!\{y\}\!\} = \{\!\{x, y\}\!\}$), and the neutral

element $\{\{\}\}$, which is an empty bag. Another type of monoid collection is the list, which can be considered an ordered bag ($List[\alpha]$ denotes a list of type α), with $\mathbb{U}_{++}$ as the unit injection function, $++$ as the associative operation and $[\,]$ as the neutral element.

Monoid algebra defines distributed operations as monoid homomorphisms over monoid collections (which represent distributed datasets). The **flatmap** operation receives a function f of type $\alpha \to Bag[\beta]$ and a collection X of type $Bag[\alpha]$ as input and returns a collection $Bag[\beta]$ resulting from the union of the results of applying f to the elements of X. This operation captures the essence of parallel processing since f can be executed in parallel on different data partitions in a distributed dataset.

The operations **groupby** and **cogroup** capture the data shuffling process by representing the reorganization and grouping of data. The **groupby** operation groups the elements of $Bag[\kappa \times \alpha]$ through the first component (key) of type κ and results in a collection $Bag[\kappa \times Bag[\alpha]]$, where the second component is a collection containing all elements of type α to which were associated with the same key k in the initial collection. The **cogroup** operation works similarly to **groupby**, but it operates on two collections that have a key of the same type κ.

The **reduce** operation represents the aggregation of the elements of $Bag[\alpha]$ into a single element of type α from the application of an associative function f of type $\alpha \to \alpha \to \alpha$. The operation **orderby** represents the transformation of a bag $Bag[\kappa \times \alpha]$ into a list $List[\kappa \times \alpha]$ ordered by the key of type κ which supports the total order $\leq$. In addition, monoid algebra also supports the use of lambda expressions ($\lambda x.e$) and **if-then-else**.

3 Data Flow

To define the DAG that represents the data flow of a data processing program, we rely on the data flow graph model presented in [6], which was formalized using Petri Nets [7]. In this model, a data flow graph is represented as a directed bipartite graph that has two types of vertices, the *links* and the *actors*. Actors are represented by the Petri net *transitions* that represent operations on data, while links are represented by the *places* that receive data from an actor and transmit data to one or more actors. Actors (transitions) and links (places) are connected through *edges* that represent communication channels between the vertices.

A program P is defined as a bipartite directed graph where places stand for the distributed datasets (D) of the program, and transitions stand for its transformations (T). Datasets and transformations are connected by edges (E):

$$P = \langle D \cup T, E \rangle$$

To exemplify the model, let us consider the Spark program shown in Figure 1. This program receives as input two datasets (RDDs) containing log messages (line 1), makes the union of these two datasets (line 2), removes duplicate logs

```
1 def unionLogsProblem(firstLogs: RDD[String], secondLogs: RDD[String])
      : RDD[String] = {
2   val aggregatedLogLines = firstLogs.union(secondLogs)
3   val uniqueLogLines = aggregatedLogLines.distinct()
4   val cleanLogLines = uniqueLogLines.filter((line: String) => !(line.
        startsWith("host") && line.contains("bytes")))
5   return cleanLogLines
6 }
```

Fig. 1: Sample log union program in Spark.

(line 3), and ends by filtering headers, removing logs which match a specific pattern (line 4) and returning the filtered RDD (line 5).

In this program we can identify five RDDs, that will be referred to by using short names for conciseness:

$$
\begin{aligned}
D &= \{d_1, d_2, d_3, d_4, d_5\} \\
d_1 &= \text{firstLogs} \\
d_2 &= \text{secondLogs} \\
d_3 &= \text{aggregatedLogLines} \\
d_4 &= \text{uniqueLogLines} \\
d_5 &= \text{cleanLogLines}
\end{aligned}
$$

We can also identify the application of three transformations in P, thus the set T in our example is defined as:

$$
\begin{aligned}
T &= \{t_1, t_2, t_3\} \\
t_1 &= \mathit{union}(d_1, d_2) \\
t_2 &= \mathit{distinct}(d_3) \\
t_3 &= \mathit{filter}(\texttt{(line: String)=>} \\
&\qquad \texttt{!(line.startsWith(“host”) \&\&} \\
&\qquad \texttt{line.contains(“bytes”))}, d_4)
\end{aligned}
$$

Each transformation in T receives one or two datasets belonging to D as input and produces a dataset also in D as output. In addition, the sets D and T are disjoint and finite:

$$D \cap T = \emptyset$$

Datasets and transformations are connected through edges. An edge may either be a pair in $D \times T$, representing an input dataset of a transformation, or a pair in $T \times D$, representing an output dataset of a transformation. In this way, the set of edges of P is defined as:

$$E \subseteq (D \times T) \cup (T \times D)$$

The set E in our example program is, then:

$$E = \{(d_1, t_1), (d_2, t_1), (t_1, d_3), (d_3, t_2), (t_2, d_4), (d_4, t_3), (t_3, d_5)\}$$

Using these sets, the DAG representing the Spark program in Figure 1 can be seen in Figure 2. The distributed datasets in D are represented as circle nodes and the transformations in T are represented as thick bar nodes og the graph. The edges are represented by arrows that connect the datasets and transformations. The token marking in d_1 and d_2 indicate that the program is ready to be executed (initial marking).

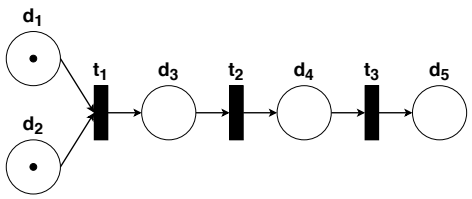

Fig. 2: Data flow representation of the program in Figure 1.

We consider two subsets of D, one that represents the input datasets (R) of a program, which are created from an external data source, and one that represents the output datasets (C) of a program, which represent datasets that will be saved or collected in some external data collector. For the sake of simplicity, we did not define in our model operations that generate datasets in R or that collect datasets in C, we assume that the vertices in these sets represent the inputs and outputs, respectively, of our program P. R and C are defined as follows:

$$R = \{d \in D | \forall t \in T.(t, d) \notin E\}$$
$$C = \{d \in D | \forall t \in T.(d, t) \notin E\}$$

Considering our example, the sets R and C are defined as:

$$R = \{d_1, d_2\}$$
$$C = \{d_5\}$$

The set of input datasets of a transformation t, and the set of output datasets of a transformation t are denoted $I(t)$ and $O(t)$:

$$I(t) = \{d \in D | (d, t) \in E\}$$
$$O(t) = \{d \in D | (t, d) \in E\}$$

Similarly, the set of transformations that gave rise to a dataset d and the set of transformations that receive a dataset d as input are denoted $I(d)$ and $O(d)$:

$$I(d) = \{t \in T | (t, d) \in E\}$$
$$O(d) = \{t \in T | (d, t) \in E\}$$

There are transformations that operate on a single dataset ($|I(t)| = 1$), which we denote *unary transformations*, and transformations that operate on two datasets ($|I(t)| = 2$), which we denote *binary transformations*. Examples of unary transformations are the *distinct* and *filter* transformations, such as the transformations applied in the program shown in Figure 1, and examples of binary transformations are the set transformations, such as the *union* in our sample program. In both types of transformations, a single dataset is always generated as an output ($|O(t)| = 1$). From this, we can derive the following properties:

$$\begin{aligned} 1 \leq |I(t)| \leq 2 \quad & \text{for all } t \in T \\ |O(t)| = 1 \quad & \text{for all } t \in T \end{aligned}$$

The datasets in D can be generated from reading an external source, which are referred as the input datasets of P that belong to the R set ($|I(d)| = 0$), or can be the result of a transformation in T ($|I(d)| = 1$). In addition, datasets in D can be input for one or more transformations in T ($|O(d)| > 1$), or they can be the output dataset of the program P , belonging to the set C ($|O(d)| = 0$). From this, we can also derive the following properties:

$$\begin{aligned} 0 \leq |I(d)| \leq 1 \quad & \text{for all } d \in D \\ |O(d)| \geq 0 \quad & \text{for all } d \in D \end{aligned}$$

Taking as a reference the transformation t_1 and the dataset d_1 of our example program P, we can see that these respect the properties presented:

$$\begin{aligned} I(t_1) &= \{d_1, d_2\} \\ O(t_1) &= \{d_3\} \\ I(d_1) &= \emptyset \\ O(d_1) &= \{t_1\} \end{aligned}$$

The data flow represented by P must be consistent with the types of the datasets in D and the transformations in T. Thus, if a dataset d is defined as input to a transformation t, then the type of d must be the same as the type of the dataset t receives as input. Likewise, if the dataset d is defined as output of t, then the type of d must be the same as the type of the dataset returned by t. To represent these properties, let's consider that binary transformations receive the two input datasets orderly. We will also consider the functions $input_type_1(t)$ of type $T \rightarrow \tau_1$, $input_type_2(t)$ of type $T \rightarrow \tau_2$, and $output_type(t)$ of type $T \rightarrow \tau_3$, to denote respectively the type τ_1 of the first input dataset of t (for the case of unary and binary transformations), the type τ_2 of the second input dataset of t (only in the case where t is a binary transformation) and the type τ_3 of the output dataset of t. In addition, let's consider the function $type(d)$,

of type $D \to \tau$, which denotes the type (τ) of a dataset. From these functions, we can define that if a dataset d is input for a unary transformation t ($(d,t) \in E \wedge |I(t)| = 1$), then the type of d must be the same as the input dataset type of t ($type(d) = input_type_1(t)$):

$$\forall t \in T, d \in D.(d,t) \in E \wedge |I(t)| = 1 \Rightarrow type(d) = input_type_1(t)$$

Similarly, if t is a binary transformation ($|I(t)| = 2$), then the type of d must be the same as the type of the first input dataset of t or equal to the type of the second input dataset of t ($type(d) = input_type_1(t) \vee type(d) = input\ _type_2(t)$):

$$\forall t \in T, d \in D.(d,t) \in E \wedge |I(t)| = 2 \Rightarrow type(d) = input_type_1(t) \vee type(d) = input_type_2(t)$$

For both types of transformations, if a dataset d is the output of a transformation t ($(t,d) \in E$), then the type of d must match the type of the output dataset of t ($type(d) = output_type(t)$):

$$\forall t \in T, d \in D.(t,d) \in E \Rightarrow type(d) = output_type(t)$$

4 Datasets and Transformations

The data flow model presented above represents the datasets and transformations of a program P, as well as the order in which these transformations are processed when P is executed. To define the semantics of datasets in D and the types of transformations in T, we make use of *Monoid Algebra* [4, 5]. Datasets are represented through monoid collections and transformations are defined as operations supported by monoid algebra. These representations are detailed in the following.

Distributed DataSets A distributed dataset in D can either be represented by a *bag* ($Bag[\alpha]$) or a *list* ($List[\alpha]$). Both structures represent collections of distributed data [5], capturing the essence of the concepts of *RDD* in Apache Spark, *PCollection* in Apache Beam, *DataSet* in Apache Flink and *DryadTable* in DryadLINQ.

We define most of the transformations of our model in terms of bags, considering lists only in the sorting transformations, which are the only ones in which the order of the elements in the dataset is relevant. In monoid algebra, bags and lists can either represent distributed or local collections. These two kind of collections are treated by monoid homomorphisms in a unified way [5]. In this way, we will not distinguish distributed and local collections when defining our transformations.

4.1 Mapping Transformations

Mapping transformations transform values of an input dataset into values of an output dataset through the application of a mapping function. Our model provides two mapping transformations: *flatMap* and *map*. Both transformations apply a given function f to every element of the input dataset to generate the output dataset, the only difference being the requirements on the type of f and its relation with the type of the generated dataset. The *map* transformation accepts any $f : \alpha \to Bag[\beta]$ and generates an output dataset of type $Bag[\beta]$, while the *flatMap* transformation requires f to directly produce $Bag[\beta]$ elements. The definition of *flatMap* in our model is just the operation **flatmap** from monoid algebra:

$$flatMap :: (\alpha \to Bag[\beta]) \to Bag[\alpha] \to Bag[\beta]$$
$$flatMap(f, D) = \textbf{flatmap}(f, D)$$

The *map* transformation derives data of type $Bag[\beta]$ when given a function $f : \alpha \to \beta$. For that to be modeled with the **flatmap** from monoid algebra, we create a lambda expression that receives an element x from the input dataset and results in a $Bag[\beta]$ collection containing only the result of applying f to x ($\lambda x.\{\!\{f(x)\}\!\}$). Thus, *map* is defined as:

$$map :: (\alpha \to \beta) \to Bag[\alpha] \to Bag[\beta]$$
$$map(f, D) = \textbf{flatmap}(\lambda x.\{\!\{f(x)\}\!\}, D)$$

4.2 Filtering Transformations

The *filter* transformation receives a function p that maps an element of the dataset to a boolean value, representing whether or not an element of the input dataset should be mapped to the output dataset. To define *filter*, we need to check whether a x element of the input dataset satisfies the condition in p ($p(x) = true$). This condition is verified in a lambda expression that uses a conditional expression (**if-then-else**) that results in $\{\!\{x\}\!\}$ when $p(x) = true$, or in $\{\!\{\}\!\}$ otherwise. This lambda expression is then applied to the input dataset using the **flatmap** operation:

$$filter :: (\alpha \to boolean) \to Bag[\alpha] \to Bag[\alpha]$$
$$filter(p, D) = \textbf{flatmap}(\lambda x.\ \textbf{if}\ p(x)\ \textbf{then}\ \{\!\{x\}\!\}\ \textbf{else}\ \{\!\{\}\!\}, D)$$

4.3 Grouping Transformations

Grouping transformations group the elements of a dataset with respect to a key. We define two grouping transformations in our model: *groupByKey* and *groupBy*. The *groupByKey* transformation is defined as the **groupBy** operation of monoid

algebra. It maps a key-value dataset into a dataset associating each key to a bag. Our *groupBy* transformation uses a function k to map elements of the collection to a key *before* grouping the elements with respect to that key:

$$\begin{aligned} groupBy &:: (\alpha \to \kappa) \to Bag[\alpha] \to Bag[\kappa \times Bag[\alpha]] \\ groupBy(k, D) &= \mathbf{groupby}(\mathbf{flatmap}(\lambda x.\{\!\{(k(x), x)\}\!\}, D)) \end{aligned}$$

$$\begin{aligned} groupByKey &:: Bag[\kappa \times \alpha] \to Bag[\kappa \times Bag[\alpha]] \\ groupByKey(D) &= \mathbf{groupby}(D) \end{aligned}$$

4.4 Set Transformations

Set transformations represent binary operations based on mathematical set operations (union, intersection and subtract). They operate on two datasets of the same type and result in a new dataset of the same type. The definition of these transformations is based on the definitions in [5].

The *union* transformation represents the union of elements from two datasets into a single dataset. This operation is represented in a simple way using the *bags* union operator ($uplus$):

$$\begin{aligned} union &:: Bag[\alpha] \to Bag[\alpha] \to Bag[\alpha] \\ union(D_x, D_y) &= D_x \uplus D_y \end{aligned}$$

Since this *union* transformation allows repeated elements, given the characteristics of the *bag* collection, unlike the union operation between mathematical sets, we defined the *distinct* transformation that removes duplicate elements from the input dataset.

To define *distinct*, we must group the elements of the dataset with the element itself as the key and then take only the key, thus, the values that are in the group, which is the repeated element itself, are discarded, thus leaving only the elements without repetition. For that, we transform the elements of the dataset into key/value elements where the key and value are the element itself. We do this by applying an operation **flatmap** ($t_1(D) = \mathbf{flatmap}(\lambda x.\{\!\{(x, x)\}\!\}, D)$). Then, we group the result dataset of this operation with the operation **groupby**, which generates a key/value set in which the key is the element itself and the value is a collection containing the repeated elements ($t_2(D) = \mathbf{groupby}(t_1(D))$). Finally, the result of this grouping is applied to a new mapping with **flatmap** that applies a function that takes the key/value element and returns only the key ($\mathbf{flatmap}(\lambda(k, g).\{\!\{k\}\!\}, t_2(D))$). Thus, *distinct* is defined as follows:

$$\begin{aligned} distinct &:: Bag[\alpha] \to Bag[\alpha] \\ distinct(D) &= \mathbf{flatmap}(\lambda(k, g).\{\!\{k\}\!\}, t_2(D)) \\ t_1(D) &= \mathbf{flatmap}(\lambda x.\{\!\{(x, x)\}\!\}, D) \\ t_2(D) &= \mathbf{groupby}(t_1(D)) \end{aligned}$$

To define the intersection and difference operations, we first define auxiliary operations *some* and *all* that represent the existential ($\exists$) and universal ($\forall$) quantifiers. These operations receive a predicate function p and reduce the dataset to a logical value. For the definition of *some* and *all*, we must first map all elements of the set to logical values by applying p to each element of D ($t_1(p, D) = \mathbf{flatmap}(\lambda x.\{\!\{p(x)\}\!\}, D)$). Having a set with only logical values, we can reduce the values of the set to a single logical value with the application of the **reduce** operation that receives a binary function and applies it associatively to the dataset.

The existential quantifier behavior is obtained from a reduction by applying the logical disjunction operator ($\vee$), which will result in true if there is at least one value x in D for which $p(x) = true$ ($\mathbf{reduce}(\vee, t_1(p, D))$). The universal quantifier ($\forall$), is obtained by reducing the dataset using the conjunction operator ($\wedge$), which is only true if all elements are true ($\mathbf{reduce}(\wedge, t_1(p, D))$). Thus, *some* and *all* are defined as:

$$
\begin{aligned}
some &:: (\alpha \to boolean) \to Bag[\alpha] \to boolean \\
some(p, D) &= \mathbf{reduce}(\vee, t_1(p, D)) \\
all &:: (\alpha \to boolean) \to Bag[\alpha] \to boolean \\
all(p, D) &= \mathbf{reduce}(\wedge, t_1(p, D)) \\
t_1(p, D) &= \mathbf{flatmap}(\lambda x.\{\!\{p(x)\}\!\}, D)
\end{aligned}
$$

Then, we define *intersection* from a mapping in D_x containing a conditional expression (**if-then-else**) that has as a condition the operation *some* in D_y passing the function of predicate that checks whether $x = y$. Thus, a x value of D_x will only enter the resulting set if the value of *some* for that x is true, which results in the intersection between D_x and D_y. The definition of *intersection* is as follows:

$$
\begin{aligned}
intersection &:: Bag[\alpha] \to Bag[\alpha] \to Bag[\alpha] \\
intersection(D_x, D_y) &= \mathbf{flatmap}(\lambda x.\ \mathbf{if}\ some(\lambda y.x = y, D_y)\ \mathbf{then}\ \{\!\{x\}\!\}\ \mathbf{else}\ \{\!\{\}\!\}), D_x)
\end{aligned}
$$

The difference (*subtract*) between two datasets D_x and D_y is defined in a similar way, however, the condition for a x element of D_x to enter the resulting set is that for every y element in D_y, the predicate $x \neq y$ is true. Therefore, we also define *subtract* with a mapping and a conditional expression, but the condition passed is the operation *all* passing the predicate function that checks whether $x \neq y$, which results in a set containing only values that are in D_x but not in D_y. The definition of *subtract* is:

$$
\begin{aligned}
subtract &:: Bag[\alpha] \to Bag[\alpha] \to Bag[\alpha] \\
subtract(D_x, D_y) &= \mathbf{flatmap}(\lambda x.\ \mathbf{if}\ all(\lambda y.x \neq y, D_y)\ \mathbf{then}\ \{\!\{x\}\!\}\ \mathbf{else}\ \{\!\{\}\!\}), D_x)
\end{aligned}
$$

4.5 Aggregation Transformations

Aggregation transformations aggregate the elements of a dataset into a single element. The most common aggregations apply binary operations on the elements of a dataset to generate a single element, resulting in a single value, or on groups of values associated with a key. We represent these aggregations with the transformations *reduce*, which operates on the whole dataset, and *reduceByKey*, which operates on values grouped by key. The *reduce* transformation has the same behavior as the **reduce** operation of monoid algebra, where an associative function f of type $\alpha \to \alpha \to \alpha$ is applied to the elements of a dataset D of type $Bag[\alpha]$ resulting in a single value of type α.

The definition of *reduceByKey* is also defined in terms of **reduce**, but since its result is the aggregation of elements associated with each key rather than the aggregation of all elements of the dataset, we first need to group the elements of the set by their keys ($\mathbf{groupby}(D)$). Then, we transform them using the operation **flatmap** which results in a key/value dataset in which the key is preserved and the value becomes the reduction of the group through the application of **reduce** with function f in the group. The *reduce* and *reduceByKey* transformations are defined as:

$$\begin{aligned} reduce &:: (\alpha \to \alpha \to \alpha) \to Bag[\alpha] \to \alpha \\ reduce(f, D) &= \mathbf{reduce}(f, D) \end{aligned}$$

$$\begin{aligned} reduceByKey &:: (\alpha \to \alpha \to \alpha) \to Bag[\kappa \times \alpha] \to Bag[\kappa \times \alpha] \\ reduceByKey(f, D) &= \mathbf{flatmap}(\lambda(k, g).\{\!\{(k, \mathbf{reduce}(f, g))\}\!\}, \mathbf{groupby}(D)) \end{aligned}$$

4.6 Join Transformations

Join transformations implement relational join operations between two datasets. We define four join transformations, which correspond to well-known join operations in relational databases: *innerJoin*, *leftOuterJoin*, *rightOuterJoin*, and *fullOuterJoin*. The *innerJoin* operation combines the elements of two datasets based on a common relationship, such as the same key. The other joins (*leftOuterJoin*, *rightOuterJoin*, and *fullOuterJoin*) allow the combination of elements from two datasets that have or not a common relationship.

Our definition of the transformation *innerJoin* was based on the definition of join presented in [3]. This transformation operates on two datasets of key/value type where the keys of both are of the same type and results in a key/value dataset where the value contains elements from both input datasets that were associated with the same key. To define *innerJoin*, which receives a dataset D_x of type $Bag[\kappa \times \alpha]$ and a dataset D_y of type $Bag[\kappa \times \beta]$ as input, we must first co-group the two datasets by applying the operation **cogroup**. This co-grouping results in a dataset containing key/value elements in which the value is two groups d_x and d_y, which have the values of D_x and D_y, respectively, which were associated with the same key k. Next, we must make mappings to associate

each k key with the combinations of the values x in d_x with the values y in d_y. This is done by applying two operations **flatmap** in a nested manner. The first **flatmap** maps the x values in d_x, so that for each x a second **flatmap** is applied that maps the y values in d_y. The result of this second mapping is a value $(k, (x, y))$ of the join. The definition of *innerJoin* is as follows:

$$
\begin{aligned}
innerJoin &:: Bag[\kappa \times \alpha] \to Bag[\kappa \times \beta] \to Bag[\kappa \times (\alpha \times \beta)] \\
innerJoin(D_x, D_y) &= \mathbf{flatmap}(\lambda(k, (d_x, d_y)).t_2(k, d_x, d_y), t_1(D_x, D_y)) \\
t_1(D_x, D_y) &= \mathbf{cogroup}(D_x, D_y) \\
t_2(k, d_x, d_y) &= \mathbf{flatmap}(\lambda x.t_3(k, x, d_y), d_x) \\
t_3(k, x, d_y) &= \mathbf{flatmap}(\lambda y.\{\!\{(k, (x, y))\}\!\}, d_y)
\end{aligned}
$$

The definition of *leftOuterJoin* is similar to the definition of *innerJoin* in its first steps, which consists of co-grouping the sets D_x and D_y and then mapping their values $(k, (d_x, d_y))$. To add the unmatched elements we use a conditional expression to verify that d_y has no elements ($d_y = \{\!\{\}\!\}$). When this is true, the result of the join is a mapping of x to d_x which results in a value $(k, (x, \{\!\{\}\!\}))$, so that the second element is represented by a empty collection ($\{\!\{\}\!\}$). If d_y is not empty, two nested **flatmap** operations are applied, similar to those done in *innerJoin*, to generate the value $(k, (x, \{\!\{y\}\!\}))$ of the join. The same steps apply to the *rightOuterJoin*, with the straightforward substitution of left and right data. The definitions of *leftOuterJoin* and *rightOuterJoin* are made as follows:

$$
\begin{aligned}
leftOuterJoin &:: Bag[\kappa \times \alpha] \to Bag[\kappa \times \beta] \to Bag[\kappa \times (\alpha \times Bag[\beta])] \\
leftOuterJoin(D_x, D_y) &= \mathbf{flatmap}(\lambda(k, (d_x, d_y)).t_2(k, d_x, d_y), t_1(D_x, D_y)) \\
t_1(D_x, D_y) &= \mathbf{cogroup}(D_x, D_y) \\
t_2(k, d_x, d_y) &= \mathbf{if}\ d_y = \{\!\{\}\!\}\ \mathbf{then}\ t_3(k, d_x)\ \mathbf{else}\ t_4(k, d_x, d_y) \\
t_3(k, d_x) &= \mathbf{flatmap}(\lambda x.\{\!\{(k, (x, \{\!\{\}\!\}))\}\!\}, d_x) \\
t_4(k, d_x, d_y) &= \mathbf{flatmap}(\lambda x.t_5(k, x, d_y), d_x) \\
t_5(k, x, d_y) &= \mathbf{flatmap}(\lambda y.\{\!\{(k, (x, \{\!\{y\}\!\}))\}\!\}, d_y)
\end{aligned}
$$

$$
\begin{aligned}
rightOuterJoin &:: Bag[\kappa \times \alpha] \to Bag[\kappa \times \beta] \to Bag[\kappa \times (Bag[\alpha] \times \beta)] \\
rightOuterJoin(D_x, D_y) &= \mathbf{flatmap}(\lambda(k, (d_x, d_y)).t_2(k, d_x, d_y), t_1(D_x, D_y)) \\
t_1(D_x, D_y) &= \mathbf{cogroup}(D_x, D_y) \\
t_2(k, d_x, d_y) &= \mathbf{if}\ d_x = \{\!\{\}\!\}\ \mathbf{then}\ t_3(k, d_y)\ \mathbf{else}\ t_4(k, d_x, d_y) \\
t_3(k, d_y) &= \mathbf{flatmap}(\lambda y.\{\!\{(k, (\{\!\{\}\!\}, y))\}\!\}, d_y) \\
t_4(k, d_x, d_y) &= \mathbf{flatmap}(\lambda x.t_5(k, x, d_y), d_x) \\
t_5(k, x, d_y) &= \mathbf{flatmap}(\lambda y.\{\!\{(k, (\{\!\{x\}\!\}, y))\}\!\}, d_y)
\end{aligned}
$$

The *fullOuterJoin* includes unmatched elements from both left and right datasets:

$$
\begin{aligned}
fullOuterJoin &:: Bag[\kappa \times \alpha] \to Bag[\kappa \times \beta] \to Bag[\kappa \times (Bag[\alpha] \times Bag[\beta])] \\
fullOuterJoin(D_x, D_y) &= \mathbf{flatmap}(\lambda(k,(d_x,d_y)).t_2(k,d_x,d_y), t_1(D_x,D_y)) \\
t_1(D_x, D_y) &= \mathbf{cogroup}(D_x, D_y) \\
t_2(k, d_x, d_y) &= \mathbf{if}\ d_x \neq \{\!\{\}\!\} \wedge d_y = \{\!\{\}\!\}\ \mathbf{then}\ t_3(k,d_x)\ \mathbf{else}\ t_4(k,d_x,d_y) \\
t_3(k, d_x) &= \mathbf{flatmap}(\lambda x.\{\!\{(k,(\{\!\{x\}\!\},\{\!\{\}\!\}))\}\!\}, d_x) \\
t_4(k, d_x, d_y) &= \mathbf{if}\ d_x = \{\!\{\}\!\} \wedge d_y \neq \{\!\{\}\!\}\ \mathbf{then}\ t_5(k,d_y)\ \mathbf{else}\ t_6(k,d_x,d_y) \\
t_5(k, d_y) &= \mathbf{flatmap}(\lambda y.\{\!\{(k,(\{\!\{\}\!\},\{\!\{y\}\!\}))\}\!\}, d_y) \\
t_6(k, d_x, d_y) &= \mathbf{flatmap}(\lambda x.t_7(k,x,d_y), d_x) \\
t_7(k, x, d_y) &= \mathbf{flatmap}(\lambda y.\{\!\{(k,(\{\!\{x\}\!\},\{\!\{y\}\!\}))\}\!\}, d_y)
\end{aligned}
$$

4.7 Sorting Transformations

Sorting transformations transform a dataset of a type that supports the total ordering *leq* into a sorted dataset, which in monoid algebra is represented by lists (*List*). Sort transformations are defined in terms of the **orderby** operation of monoid algebra, which transforms a $Bag[\kappa \times \alpha]$ into a $List[\kappa \times \alpha]$ ordered by the key of type κ that supports the total order $\leq$. We will also consider the *inv* function, which transforms the total order of an element from $\leq$ to $\geq$. In addition, in monoid algebra the operation textbfflatmap preserves the order of elements in a list, while the operations **groupby** and **cogroup** not.

We define two sorting transformations, the transformation *orderBy*, which sorts a dataset $Bag[\alpha]$ where the type α supports a total order $\leq$, and the transformation *orderByKey*, which sorts a dataset of type $Bag[\kappa \times \alpha]$ where *kappa* supports the total order $\leq$. In both transformations, it is possible to indicate whether the sorting is done in an ascending ($\leq$, default), or descending ($\geq$) order. This control is carried out by a conditional expression (**if-then-else**). If the ordering is descending, the application of *inv* during the transformation of the dataset into a key/value pattern is responsible for the order inversion. The result of the conditional expression, which is a dataset of type $Bag[\alpha \times \alpha]$, is then applied to the operation **orderby**, which results in a list of type $List[\alpha \times \alpha]$ ordered by key. Finally, we make a mapping in the list to get the dataset back to its original type α. Since the **flatmap** operation maintains the order of a list, the resulting list is ordered as well. The definition of *orderBy* is as follows:

$$
\begin{aligned}
orderBy &:: boolean \to Bag[\alpha] \to List[\alpha] \\
orderBy(desc, D) &= \mathbf{flatmap}(\lambda(k,v).[k], \mathbf{orderby}(t_1(desc, D))) \\
t_1(desc, D) &= \mathbf{if}\ desc\ \mathbf{then}\ t_2(D)\ \mathbf{else}\ t_3(D) \\
t_2(D) &= \mathbf{flatmap}(\lambda x.\{\!\{(inv(x), x)\}\!\}, D) \\
t_3(D) &= \mathbf{flatmap}(\lambda x.\{\!\{(x, x)\}\!\}, D)
\end{aligned}
$$

The definition of the *orderByKey* transformation is simpler, since *orderByKey* operates on a key/value dataset. The only mapping done is when the ordering is done in descending order, so it is necessary to make a mapping to apply inv to the keys of type κ to change the total order to $\geq$. Thus, the transformation *orderByKey* is defined as follows:

$$
\begin{aligned}
orderByKey &:: boolean \rightarrow Bag[\kappa \times \alpha] \rightarrow List[\kappa \times \alpha] \\
orderByKey(desc, D) &= \mathbf{orderby}(t_1(desc, D)) \\
t_1(desc, D) &= \mathbf{if}\ desc\ \mathbf{then}\ t_2(D)\ \mathbf{else}\ D \\
t_2(D) &= \mathbf{flatmap}(\lambda(k, x).\{\!\{(inv(k), x)\}\!\}, D)
\end{aligned}
$$

5 Conclusions

This report presented the definition of a model to represent programs for large scale data processing. The model was developed from common features found in the programming models of the systems *Apache Spark*, *Apache Flink*, *Apache Beam* and *DryadLINQ*. With that, we defined a program from a data flow graph composed of data sets and transformations in order to simulate the representation of DAG adopted by these systems. The transformations were also formalized in order to represent the types of operations prevalent in these systems. Therefore, our model was defined in order to represent the main functional characteristics of programs for data processing, allowing a program to be designed and analyzed independently of the system that will execute it.

References

1. Beam, A.: Apache Beam: An advanced unified programming model (2016), `https://beam.apache.org/`
2. Carbone, P., Ewen, S., Haridi, S., Katsifodimos, A., Markl, V., Tzoumas, K.: Apache Flink: Stream and Batch Processing in a Single Engine. IEEE Data Engineering Bulletin **38**(4), 28–38 (2015)
3. Chlyah, S., Gesbert, N., Genevès, P., Layaïda, N.: An Algebra with a Fixpoint Operator for Distributed Data Collections (Mar 2019), `https://hal.inria.fr/hal-02066649`, working paper or preprint
4. FEGARAS, L.: An algebra for distributed big data analytics. Journal of Functional Programming **27**, e27 (2017). https://doi.org/10.1017/S0956796817000193
5. Fegaras, L.: Compile-time query optimization for big data analytics. Open Journal of Big Data (OJBD) **5**(1), 35–61 (2019), `https://www.ronpub.com/ojbd/OJBD_2019v5i1n02_Fegaras.html`
6. Kavi, K.M., Buckles, B.P., Bhat, N.: A Formal Definition of Data Flow Graph Models. IEEE Transactions on Computers **C-35**(11), 940–948 (Nov 1986). https://doi.org/10.1109/TC.1986.1676696
7. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (April 1989). https://doi.org/10.1109/5.24143

8. Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, Universität Hamburg (1962)
9. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P.K., Currey, J.: Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. pp. 1–14. OSDI'08, USENIX Association, Berkeley, CA, USA (2008), http://dl.acm.org/citation.cfm?id=1855741.1855742
10. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster Computing with Working Sets. In: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing. pp. 10–10. HotCloud'10, USENIX Association, Berkeley, CA, USA (2010), http://dl.acm.org/citation.cfm?id=1863103.1863113